Grundlagen der Informatik III

Wintersemester 2013/2014

Prof. Dr.-Ing. Michael Goesele,
Simon Fuhrmann, Fabian Langguth

TECHNISCHE UNIVERSITÄT DARMSTADT

2. Praktikum

06.11.2013

Das Praktikum wird in Teams von drei Personen bearbeitet, die alle der selben Übungsgruppe angehören müssen. Vermerken Sie im Quellcode alle Teammitglieder mit Matrikelnummern. Die Lösung muss bis **spätestens 27.11.** im Moodle hochgeladen werden. Es genügt, wenn *ein* Teammitglied die Lösung abgibt. Den eigentlichen Testattermin vereinbaren Sie selbst mit Ihrem Tutor.

Geben Sie alle zum Kompilieren und Ausführen Ihres Programms benötigten Dateien in einem .zip- oder .tar.gz-Archiv ab. Ihre Abgabe muss mit den auf dem Aufgabenblatt angegebenen Anweisungen auf den RBG-Poolrechnern kompilier- und ausführbar sein, ansonsten wird sie mit 0 Punkten bewertet.

Wenn Sie dieses Praktikum erfolgreich bearbeiten und die Klausur bestehen, erhalten Sie **zwei zusätzliche Klausurpunkte**.

Am 14.11.2013 sowie am 21.11.2013 zwischen 16:15 Uhr und 17:30 Uhr finden im C-Pool in S2|02 Sprechstunden statt, in der Fragen bzgl. dieses Praktikums geklärt werden können.

MandelBrot – Komplexe Zahlen visualisieren

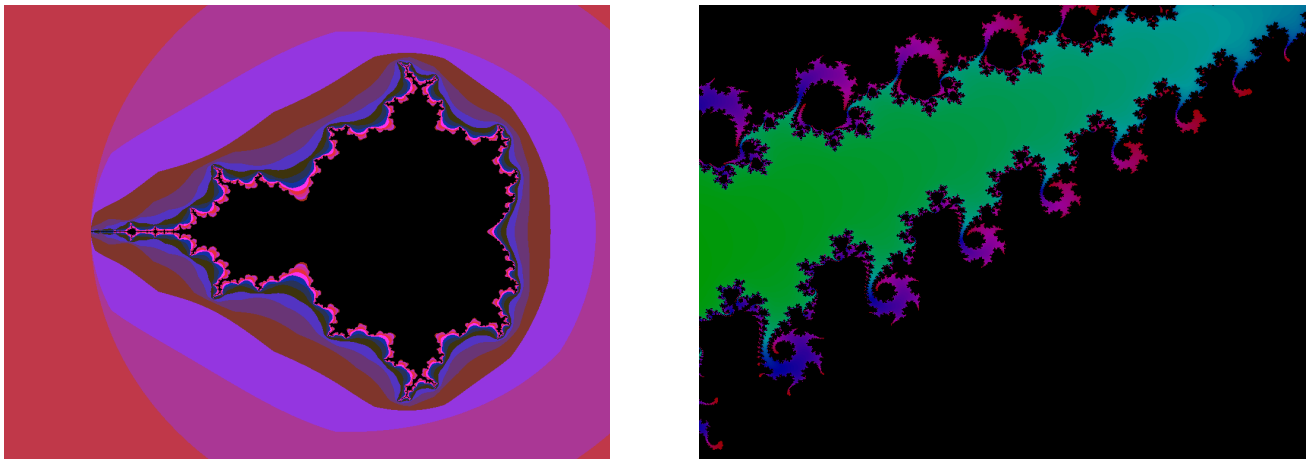

Abbildung 1: Mandelbrot-Menge in verschiedenen Farbschemata (YUV und HSV) und Zoomstufen

In der Vorlesung *Mathematik I für Informatiker* haben Sie die komplexen Zahlen als einen Körper kennen gelernt, auf dem sich Operationen wie Addition und Multiplikation definieren lassen. In diesem Praktikum nutzen Sie Ihr Wissen, um fraktale (selbstähnliche) Bilder der "Mandelbrotmenge" zu erzeugen. Erinnern Sie sich zunächst, dass sich eine komplexe Zahl $z \in \mathbb{C}$ ausdrücken lässt als $z = a + bi,\ a, b \in \mathbb{R}$. Wir betrachten nun die Folge

$$z_{n+1} = z_n^2 + c,\ z, c \in \mathbb{C}$$

mit $z_0 = 0$. Die Mandelbrotmenge ist nun definiert als Menge aller $c \in \mathbb{C}$, für die die gegebene Folge beschränkt ist, d.h.

$$\mathcal{M} = \{c \in \mathbb{C} \mid \exists b \in \mathbb{R}\ \forall n \in \mathbb{N} : |z_n| \leq b\}.$$

Ihre Aufgabe ist nun, diese Menge in einem Bild zu visualisieren. Dazu sollen Sie den an "Normalized Iteration Count" angelehnten Algorithmus implementieren. In der Praxis setzen Sie dazu zunächst eine maximale Anzahl an Iterationen $n_{\max}$ sowie einen Radius $b = r_{\max}$ fest. Sie betrachten nun einen Auschnitt der komplexen Zahlenebene

und ordnen jedem Pixel des Ausgabebildes gleichabständig eine komplexe Zahl $c \in \mathbb{C}$ zu. Für jeden Pixel iterieren Sie nun die o.g. Folge für maximal $n_{\max}$ Iterationen. Die Folge gilt genau dann als beschränkt, wenn für alle $n \leq n_{\max}$ gilt, dass $|z_n| \leq r_{\max}$. Pixel mit beschränkten Folgen werden abschließend schwarz ausgegeben. Für alle anderen Pixel speichern Sie die Anzahl der Iterationen, in denen der Kreis mit dem Radius $r_{\max}$ nicht verlassen wurde – also das größte n sodass $|z_n| \leq r_{\max}$. Diese Information wird anschließend transformiert und durch Color-Mapping einer Farbe zugeordnet. Im Pseudo-Code sieht der Algorithmus folgendermaßen aus:

```
z = 0 + 0i
c = getComplexValueForPixel(x,y)
iteration = 0

while (abs(z) <= rMax and iteration < maxIteration) {
   z = z*z + c
   iteration++
}

if (iteration < maxIteration) {
   mu = log( log(abs(z)) / log(2)) / log(2)
   iteration = iteration + 1 - mu
}

color = colorMap(iteration)
setPixelColor(x,y,color)
```

(vgl. `http://en.wikipedia.org/wiki/Mandelbrot_Set#Continuous_.28smooth.29_coloring`).

Für das Color-Mapping wird üblicherweise eine fixe Palette genutzt. Für dieses Praktikum setzen Sie eine einfache Variante um, in der der YUV-Farbraum ausgenutzt wird. Im Pseudocode ist das Schema:

```
Y = 0.2
U = -1 + 2 * (iterations / maxIterations)
V = 0.5 - (iterations / maxIterations)
colorRGB = convertYUVToRGB(Y,U,V)
```

Die passende Konvertierungsfunktion nach RGB finden Sie bei Wikipedia (Artikel “YUV”).

Sie erhalten für dieses Praktikum sowohl ein Kommandozeilenprogramm als auch eine GUI (siehe Abschnitt “GtkBrot”). Ihre Aufgabe ist es nun, in der Datei `Mandelbrot.c` obigen Algorithmus sowie eine Color-Mapping-Funktion zu implementieren. Ein dokumentiertes Interface finden Sie in `Mandelbrot.h`. Sobald Sie den Code vervollständigt haben, können Sie das Programm mit `make cli` kompilieren. Mit `./mandelbrot_cli` (`-help` für Parametererklärung) wird dann ein Bild mit $n_{\max} = 100$ gerendert und als `Mandelbrot.ppm` abgespeichert. Wenn Ihr Programm korrekt arbeitet, erhalten sie ein Bild (YUV-Schema) wie in Abbildung 1.1. Als maximalen Radius können Sie die Konstante `RADIUS` aus `globals.h` nutzen.

FastBrot – SSE in Aktion

Die Funktion `testEscapeSeriesForPoint` arbeitet bisher pro Aufruf an einem Pixel. Mittels SSE können Sie – wie in der Vorlesung vorgestellt – die Datenparallelität der Aufgabe ausnutzen und die Funktion vektorisieren. Sie müssen dabei das gegebene Interface verändern und an Ihre Bedürfnisse anpassen (außer der Funktion `generateMandelbrot`).

Dabei sollen Sie die sogenannten *Intrinsics* verwenden. Das sind Funktionen, die den komfortablen Zugriff auf SSE-Funktionen unter C ermöglichen. Binden Sie dazu die folgenden Header ein:

```
#include <xmmintrin.h> // SSE 1
#include <emmintrin.h> // SSE 2
#include <pmmintrin.h> // SSE 3
```

Ein SSE-Register entspricht dem neuen Datentyp `__m128`. Für nahezu jeden Assembler-SSE Opcode gibt es eine Entsprechung als Intrinsic – eine Referenz finden Sie auf [1] (SSE, SSE2 und SSE3) sowie in dem im Moodle

bereitgestellten PDF-Dokument von Intel. Eine kurze Einführung finden Sie unter [2]. Ihre Aufgabe ist es nun, Ihre serielle Mandelbrot-Implementierung sowie die YUV-Color-Mapping-Funktion aus der letzten Aufgabe mittels SSE zu vektorisieren.

Wie Sie mit `cat /proc/cpuinfo` feststellen können, unterstützen die Pool-Rechner sowohl SSE1 und SSE2 als auch SSE3. Für den maximalen Speedup empfehlen wir Ihnen, die SSE3-Intrinsics `_mm_move*dup_ps` und `_mm_addsub_ps` zu studieren.

1: `http://msdn.microsoft.com/en-us/library/y0dh78ez%28v=vs.90%29.aspx`
2: `http://supercomputingblog.com/optimization/getting-started-with-sse-programming/`

GtkBrot – Erkunden von fraktalen Mengen

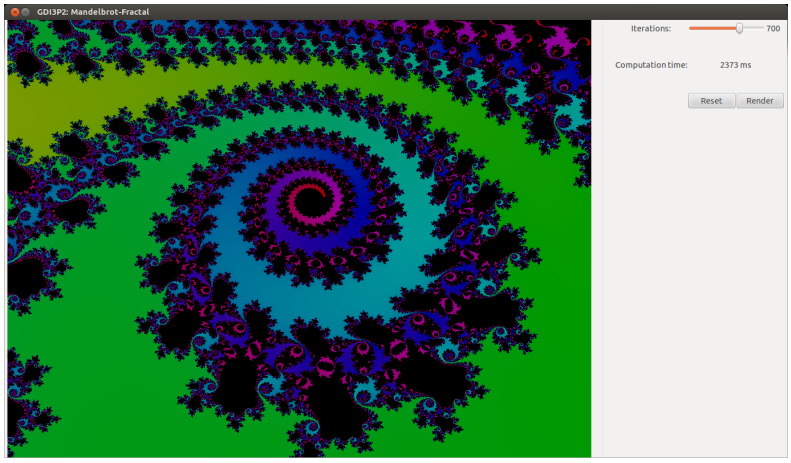

Abbildung 2: Bereitgestellte GTK-GUI für Ihren Mandelbrot-Algorithmus

Bei Nutzung des Konsolenprogramms bekommen Sie leider nur ein Bild von der gesamten Mandelbrot-Menge. Zum Erkunden der fraktalen Menge stellen wir eine auf GTK+ basierende GUI bereit. Durch Klick in das entstehende Bild können Sie bis zu 16 Mal in die Menge hineinzoomen. Die erforderlichen Entwicklungspakete sind bereits auf den Poolrechnern des FB Informatik installiert. Mit `make gui` können Sie die GUI kompilieren und mit `./mandelbrot_gui` ausführen.

Arbeiten Sie auf Ihrem eigenen Rechner, so benötigen Sie die GTK+-2.0-Entwicklungspakete (unter Ubuntu etwa `libgtk2.0-dev`).

Hinweis: Die Nutzung der GUI legen wir Ihnen zur Erkundung der Mandelbrotmenge ans Herz. Sie ist jedoch keine geforderte Leistung im Rahmen des Praktikums, d.h. sie dient Ihnen nur zur Kontrolle.

Bearbeitungshinweise

Analog zum letzten Praktikum werden alle Arrays eindimensional angelegt. Der Pixel `(x,y,z)` in einem 3-Kanal-Bild lässt sich mit `image[y*width*3 + x*3 + z]` (”row-major Format”) ansprechen. Im gegebenen Interface finden Sie Informationen über die erwarteten Arraylängen und -formate. Der Standard C99 erlaubt direkte Interaktion mit komplexen Zahlen über den Header `complex.h` (und enthaltene Funktionen) und den Datentyp `complex float`. Das Interface übergibt Ihnen Argumente standardmäßig in diesem Format. Für die eigentliche Berechnung raten wir jedoch von der Benutzung der eingebauten Multiplikation und Addition ab. Wir empfehlen eine eigene Implementierung der Addition und Multiplikation, durch welche sich Geschwindigkeitsunterschiede bis zu einem Faktor von 8 ergeben. Mit den Funktionen `crealf(c)` und `cimagf(c)` haben Sie Zugriff auf den Real- und Imaginärteil der komplexen Zahl `c`.

Bewertung

In diesem Praktikum können sie bis zu zwei ganze Punkte erreichen. Die Punktevergabe ergibt sich wie folgt, wobei für das Bestehen mindestens der erste ganze Punkt nötig ist:

- Sie erhalten einen *ganzen* Punkt für eine korrekte, serielle Implementierung des o.g. Algorithmus inkl. des YUV-Colormappings, sowie eine korrekte Implementierung des o.g. Algorithmus und des YUV-Colormappings in SSE. (Der SSE Algorithmus muss dabei das gleiche Ergebnis der seriellen Implementierung liefern.)
- Sie erhalten einen *halben* Punkt, wenn Ihre korrekte SSE-Implementierung mindestens einen Speedup von `1.5x` gegenüber der seriellen Variante (siehe Hinweis zur `complex`-Arithmetik) aufweist.
- Sie erhalten einen *weiteren halben* Punkt, wenn Ihre korrekte SSE-Implementierung mindestens einen Speedup von `1.8x` gegenüber der seriellen Variante (siehe Hinweis zur `complex`-Arithmetik) aufweist.

Um den Speedup zu messen, lassen wir `mandelbrot_cli` ein Bild der Auflösung `1024 x 768 px` mit 10000 Iterationen rendern. **Die Tutoren werden Ihre serielle Implementierung genau prüfen. Falls diese künstlich verlangsamt wurde, können Sie keinen Speedup-Punkt mehr erhalten!** Verwendet Ihre serielle Implementierung die `complex.h`-Arithmetik, so messen wir gegen eine Referenzimplementierung.

Insbesondere erhalten Sie *keine* Punkte, wenn Ihr Programm nicht ohne Veränderung auf den RBG-Poolrechnern kompilierbar und lauffähig ist. Testen Sie Ihr Programm daher *unbedingt* vor der Abgabe. Achten Sie außerdem – auch in Ihrem eigenen Interesse – auf lesbaren, gut kommentierten Code. Sie werden im Testat Code-Ausschnitte erklären müssen.

Wir wünschen Ihnen viel Spaß beim Bearbeiten des Praktikums und freuen uns über die Einsendung interessanter und kreativer Bilder!